VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

Diplomová práce

2011 Roman Konkol

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vykreslování flowgrafu pro provádení procesu popsaných formalismem Pi-kalkulu

Creating Flowgraph Chains Based on the Pi-calculus Formalised Processes

2011 Roman Konkol

ZADÁNÍ

Poděkování

Rád bych poděkoval vedoucímu své diplomové práce Ing. Štěpánu Kuchařovi, který mi poradil mnoha cennými radami a v nemálo situacích popostrčil správným směrem během tvorby bakalářské a hlavně této diplomové práce.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 5.5.2011 Roman Konkol

Abstrakt

Cílem této práce je vytvořit systém schopný zobrazit komunikaci mezi procesy popsané algebrou Pi-kalkulu pomocí flowgrafů a navrhnout vhodné uložení formálního zápisu procesů v Pi-kalkulu.

V rámci této práce jsem popsal formalismus Pi-kalkulu, postup převodu formalismu do XML dokumentu a navrhl systém zobrazující Pi-kakul pomocí flowgrafů.

Klíčová slova

pi-kalkul, flowgraf, XML, Extensible Markup Language, procesní kalkul, proces

Abstract

The main objective of this thesis is to create an application that's capable to represent communicating processes described by Pi-calculus with flowgraphs and suggest suitable deposition for Pi-calculus formalism.

In this thesis I described a transfer plan of Pi-calculus formalism to XML document and I suggest an application portrayed Pi-calculus processes by flowgraphs.

Keywords

Pi-calculus, flowgraph, XML, Extensible Markup Language, process calculus, process

Slovník, použitá slova

XML - eXtensible Markup Language

UML - Unified Modeling Language

DTO - Data Transfer Object

DOM - Document Object Model

SAX - Simple API for XML

Obsah

Seznam obrázků

1. Úvod

Proces je obecným označením pro děje, které se postupně mění v čase a mění své chování a stav. Mnoho systémů popisuje chování jednotlivých procesů, jak fungují, proč vytvářejí nebo ukončují své funkce a podobně. Málokterý popisuje chování a komunikaci mezi těmito procesy, právě Pi-kalkul je touto vyjímkou a na konkrétní funkce procesů pracuje jako s černou skříňkou.

Pi-kalkul je matematický formalismus sloužící k popisu a analýze souběžných procesů. Jinak řečeno, pomocí pi-kalkulu jsme schopni formálně popsat komunikaci v procesech i mezi procesy. Pi-kalkul má jednu velkou nevýhodu, ve své prosté formální podobě je málo názorný a proto se k jeho zobrazení používají takzvané flowgrafy. Tyto flowgrafy bohužel nejsou vhodné pro ruční zobrazování procesů pi-kalkulu, protože se při každé změně procesu musí vykreslovat celé znovu.

Tato práce se bude zabývat v kapitole 2. Pi-kalkul formálním zápisem p-kalkulu a zobrazováním pomocí flowgrafů. 3 Často používané nástroje je kapitolou, ve které popisuji formální definici nástrojů, které jsem ve své práci často využíval. V kapitole 4. Pi-kalkul elektronicky popíšu možné způsoby uložení procesů pi-kalkulu v elektronické podobě a aktuální programy převádějící pi-kalkul do grafické podoby. 5.návrh programu obsahuje vytvářením systému, který je schopen tyto způsoby uložení zobrazit pomocí flowgrafů. V poslední kapitole 6. Závěr jsem zhodnotil celý průběh tvorby diplomové práce.

2. Pi-kalkul

Tato kapitola pojednává o formalismu pi-kalkulu, popisuje zápis, chování a čerpá v převážné části z knihy Komunikační a mobilní systémy viz. [1], doplněné o informace z kapitoly věnující se pi-kalkul[2] a článku [3]. Pro základní představu k pi-kalkulu jsem nahlédl na krátký článek [4] nebo zajímavé video přednášky [5]

První publikaci o pi-kalkulu vytvořil Robin Milner, Joachim Parrow a David Walker v návaznosti na procesní kalkul CCS (Calculus of Communicationg Systems) na konci osmdesátých let, publikována byla roku 1992 s názvem „A calculus of mobile processes". Ze samotného názvu lze odpozorovat, že mobilita je základem pi-kalkulu. Ovšem slovo mobilita obsahuje v dnešní době mnohem více, než před lety, což je způsobeno hlavně obrovským rozvojem síťových technologií a internetem.

Pi-kalkul je matematický formalismus sloužící k popisu a analýze souběžných procesů. Pi-kalkul může hrát dvě rozdílné role.

Modeluje vytváření sítě v moderním smyslu, ve kterém jsou zprávy posílány z místa na místo. Tyto zprávy mohou obsahovat vazby na jiné aktivní procesy nebo mohou dokonce obsahovat procesy samotné.

Zadruhé se můžeme dívat na pi-kalkul jako na základní výpočetní model. Každý základní model spočívá na malém množství primitivních výrazů. Pi-kalkul spočívá na primitivních výrazech interakce, stejně jako Turingovy stroje spočívají na výrazech čtení a zápisu na paměťové médium.

2.1. Mobilita

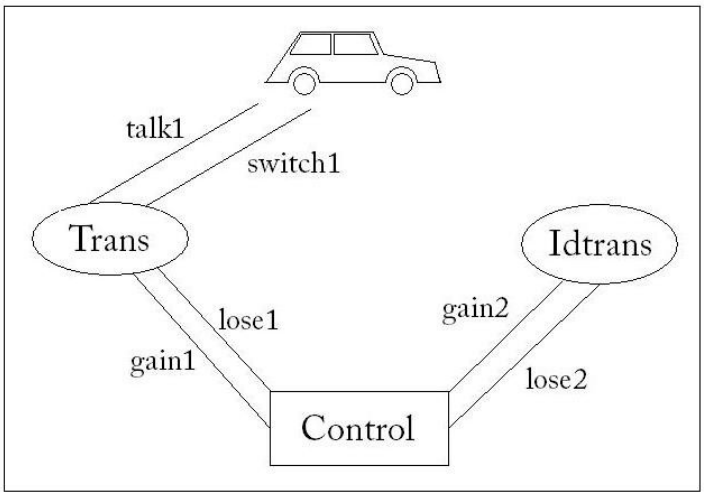
Mobilita u pi-kalkulu je schopnost změny na základě meziprocesní komunikace. Komunikující procesy jsou vzájemně spojeny komunikačním kanálem, tyto kanály se různě přesouvají, vznikají a zanikají. Vnitroprocesní komunikace není z vnějšího pohledu viditelná. Mobilitu se pokusím blíže ukázat na příkladu.

Rektor VŠB-TU Ostrava jede automobilem konzultovat spojení s Ostravskou univerzitou, během jízdy ze svého automobilu přes handsfree telefonuje se svou sekretářkou. Telefonní signál je přenášen z automobilu anténou a dál do řídícího centra, odkud je přesměrován do kanceláře sekretářky. Je zde, ale jeden problém, anténa má omezený dosah příjmu, který nedokáže pokrýt celou oblast, kterou automobil projíždí. Mobilní signál musí být přesouván postupně mezi více anténami, aby spolu mohli volající komunikovat. Změnu příjmu řídí řídící středisko přiřazováním telefonu automobilu jednotlivým anténám. Procesy zde zastupují telefon v automobilu, antény, řídící centrum, telefon sekretářky. Telefon je stále spojen s právě jednou anténou, řídící centrum je stále spojeno se všemi anténami a telefonem sekretářky, řídící centrum odebírá a přidává jednotlivým anténám odkaz

na automobil.

Tento příklad se pokusím přiblížit podrobněji s jedním zjednodušením, budu vysvětlovat pouze přesun spojení mezi anténami, spojení mezi řídícím centrem a sekretářkou neberu v potaz.

V tomto příkladu jsou přítomny čtyři procesy - automobil, anténa přenášející signál Trans, řídící centrum Control a anténa, u které si přejeme, aby převzala signál z automobilu.



Ilustrace 1: Mobilita - přenos telefonního signálu 1

Automobil může hovořit přes anténu pomocí portu talk, ale v kteroukoliv chvíli může být anténa Trans upozorněna na nutnost změny antény řídícím centrem za pomoci signálu přes lose1. V tom případě pošle anténa zprávu o změně automobilu prostřednictvím switch a dál již nepřenáší zprávy automobilu a funguje jako nevyužívaná anténa Idtrans.

Trans(talk, switch, gain, lose) = talk . Trans(talk, switch, gain, lose) + lose(t, s) . switch<t, s> . Idtrans(gain, lose)

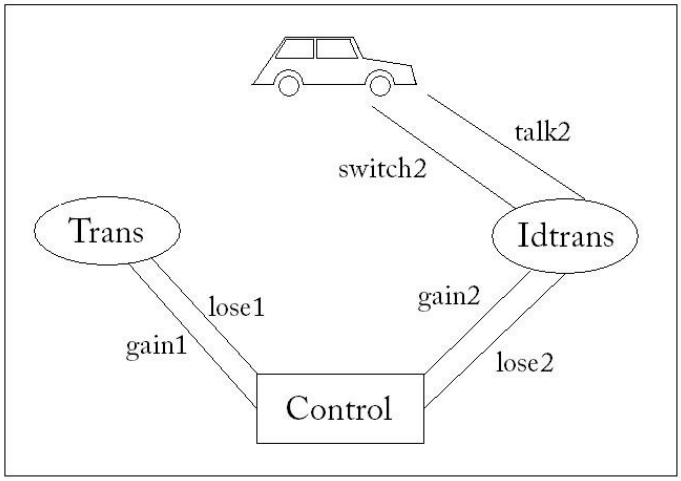
Car(talk, switch) = talk<> . Car(talk, switch) + switch(t, s). Car(t, s)

Idtrans(gain,lose) = gain(t,s).Trans(t, s, gain, lose)

Control1 = lose1<talk2,switch2> . gain2<talk2,switch2> Control2

Control2 = lose2<talk1,switch1> . gain1<talk1,switch1> Control1

Automobil stále komunikuje přes talk a opět pokračuje nezměněným procesem Car. Jednou za čas dostane automobil signál přes switch, ve kterém je upozorněn s jakými porty má dál komunikovat a pokračuje v telefonování.



Ilustrace 2: Mobilita - přenos telefonního signálu 2

Na začátku nevyužívaná anténa Idtrans čeká na přijetí zprávy od řídícího centra s informací, který hovor má přenášet (t, s), dál pokračuje jako anténa přenášející telefonní hovor Trans.

Řídící centrum jen rozhoduje pomocí signálu lose1 nebo lose2, která anténa ukončí spojení s automobilem a která následně bude komunikovat s automobilem pomocí předaných informací talk a switch.

2.2. Formální definice pi-kalkulu

Pi-kalkul je algebrou procesů, tedy matematicky definuje proces a zavádí operace s těmito procesy. V pi-kalkulu jsou procesy popsány jedinečným názvem, seznamem portů příslušného procesu a pravidly, kterými se proces řídí.

$$Semafor(barva) = \overline{barva}\langle červená \rangle + \overline{barva}\langle žlutá \rangle + \overline{barva}\langle zelená \rangle$$

Označení procesů v pi-kalkulu začíná obecně velkým písmenem (Semafor), porty procesů začínají malými písmeny $(barva\ nebo\ \overline{barva})$, v tomto případě se navíc jedná o jeden a ten samý port, bez nadtržení značí příjem zprávy přes barva a s nadtržením odeslání zprávy přes port barva. Obsah zpráv se zobrazuje pomocí kulatých závorek (zpráva) u příjmu zprávy a <zpráva> u odeslání zprávy. Zpráv může být najednou odesláno i více, jsou odděleny čárkou.

2.2.1. Základní operace pi-kalkulu

0 neprovede se žádná operace.

τ nesledovatelná akce (akce, která se provede, ale není z vnějšího pohledu viditelná).

$\pi . P$ sekvence, je operací, ve které se postupně provádějí akce jak jdou za sebou, není možné některou akci vynechat. Jednotlivé akce sekvence jsou odděleny tečkou.

new g , e P operace začínající klíčovým slovem new vytváří nové zprávy procesu. Za klíčovým slovem new v řetězci následuje seznam vytvářených zpráv oddělených čárkou a poslední informací v operaci je podproces, pro který jsou zprávy vytvořeny. V tomto příkladu jsou zprávami g a e, pro podproces P.

$P_1 + P_2$ označuje nedeterministický výběr. Jedná se o volbu, buď zvolím P_1 nebo P_2 , není možné spustit oba podprocesy.

$P_1 | P_2$ je souběžné provádění procesů P_1 a P_2. Tyto procesy jsou aktivní oba současně a mohou mezi sebou navzájem komunikovat posíláním nebo přijímáním zpráv.

$! P$ označuje replikaci procesu P, který může běžet až nekonečně mnohokrát současně. Operaci replikace je možné zapsat rozepsat do tvaru souběžného spuštění procesu P a jeho replikace !P. $! P = P | ! P$

Množina výrazů procesů pi-kalkulu je definována tímto způsobem:

$P ::= Q \mid P_1 | P_2 \mid new\ g\ P \mid ! P$
$Q ::= P_1 + P_2 \mid \pi . P \mid 0$

2.3. Zápis procesů pi-kalkulu

V této práci používám jediný zápis pi-kalkulu, ve tvaru skládajícím se ze dvou částí, definice procesu a definice chování procesu, části jsou odděleny znakem =.

Definice procesu začíná názvem procesu, za kterým jsou definovány v závorce počáteční názvy portů. Popis chování procesu je popsán složením operandů a operací. Operandy jsou odesílání či příjem zpráv označených pomocí názvů portů a kulatých nebo ostrých závorek, podprocesy s nebo bez definice počátečních portů a vytvářené zprávy za klíčovým slovem new.

Pro lepší představu o zápisu uvádím jeden kompletní příklad na objednávku

automobilu zákazníkem, ve kterém jsou použity všechny operace kromě replikace. První proces Vše spouští paralelně všechny procesy Zákazník, Banka, Sklad, Výroba a Prodejce. Zákazník představuje zákazníka, který ví kolik má peněz na účtě, kde je prodejna automobilů a jaký chce automobil jehož specifikaci vytvoří. Prodejce má přehled o své prodejně, bance, do které mu zákazníci posílají peníze, o skladu obsahujícím vyrobené automobily, a když nemá automobil na skladě ví, že se může obrátit na sklad. Obdobným způsobem se dá vysvětlit i proces Banka, Sklad, Výroba.

Vše = ZÁKAZNÍK (prodejna, účet) | BANKA (účet, banka) | SKLAD (na_skladě) | VÝROBA (ve_výrobě) | PRODEJCE (prodejna, banka, na_skladě, ve_výrobě)

ZÁKAZNÍK(prodejna, účet) = new specif (prodejna<specif> . prodejna(účet) . účet<platba> . prodejna(auto))

BANKA(účet, banka) = účet(platba).banka<zaplaceno>

SKLAD(na_skladě) = na_skladě(specif).na_skladě<auto>

VÝROBA(ve_výrobě) = ve_výrobě(specif).new auto(ve_výrobě<auto>)

PRODEJCE(prodejna, banka, na_skladě, ve_výrobě) = prodejna(specif).prodejna<účet>.(banka(zaplaceno)| ((na_skladě<specif>.na_skladě(auto))+(ve_výrobě<specif> .ve_výrobě(auto)))). prodejna<auto>

2.4. Reakční pravidla

$$TAU: t.P + M \rightarrow P$$

$$REACT: (x(y).P + M) \mid (\bar{x}<z>.Q + N) \rightarrow \{z/y\}P \mid Q$$

$$PAR: \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$RES: \frac{P \rightarrow P'}{new\ x\ P \rightarrow new\ x\ P'}$$

$$STRUCT: \frac{P \rightarrow P'}{Q \rightarrow Q'}, když\ P \equiv Q \quad a \quad P' \equiv Q'$$

Tau pravidlo popisuje vybrání jednoho prvku z operace výběru. Ve výběru jsou na počátku dva prvky t.P a M. Po provedení výběru t.P s provedením nesledovatelné akce t zůstane k provedení pouze proces P.

React popisuje reakci procesů na současné spuštění odeslání a příjmu zpráv přes stejný port. Část $(x(y).P + M)$ je výběr, ve kterém vybereme $x(z).P$, nyní čekáme na příjem zprávy **z** přes port x a po jejím přijetí spustíme proces P. S prvním podprocesem běží současně druhý podproces $(\bar{x}<z>.Q + N)$, u kterého vybereme první část kde odesíláme zprávu **z** po portu **x** a pokračujeme procesem Q.

$\{z/y\}P|Q$ popisuje situaci po odeslání a přijetí zprávy. Protože jsme přes port **x** odesílali zprávu **z** a přijali zprávu **y**, přepíšeme všechny výskyty **y** v procesu P za **z** a proces pokračuje současně spuštěnými procesy P a Q.

Zprávy jsou odesílány a přijímány vždy v párech po portech se stejnými názvy, není možné poslat více zpráv po různých portech a následně všechny přijmout. Po přijetí zprávy přes port záleží na pořadí a obsahu ostrých i kulatých závorek odesílání a přijímání zprávy. Pokud jsou kulaté závorky prázdné, jsou vytvořeny nové zprávy. Když kulaté závorky mají nějaký obsah, je tento obsah přepisován na základě reakčního pravidla v přesném pořadí jak zprávy přišly.

Pravidla Par a Res popisují shodu v situacích, ve kterých po provedení jakýchkoliv kroků z procesu P se dostaneme do procesu P' a při shodném procesu P běží zároveň proces Q a po shodném provedení stejných kroků se dostaneme do $P'|Q$ obdobně to platí i pro new x.

Struct popisuje ekvivalenci výsledných procesů P' a Q' po provedení stejné akce ekvivalentních procesů P a Q s rozdílnými názvy.

2.5. Strukturální shoda

Shoda dvou procesů v pi-kalkulu je, když

1. můžeme změnit vázané názvy
2. $P+Q+R \equiv Q+P+R \equiv P+R+Q$
3. $P|0 \equiv P,\ P|Q \equiv Q|P,\ P|(Q|R) \equiv (P|Q)|R$
4. $new\ x\,(P|Q) \equiv P|new\ x\ Q\ if\ x \notin fn(P)$

   $new\ x0 \equiv 0, new\ x, y\ P \equiv new\ y, x\ P$
5. $!P \equiv P|!P$

První pravidlo zajišťuje jedinečnost názvů zpráv a portů v procesech. Například v $P = new\ a((a.Q_1 + b.Q_2) \mid a\langle\rangle) \mid (b\langle\rangle.R_1 + a\langle\rangle.R_2)$ se snažím vytvořit novou zprávu a v $((a.Q_1 + b.Q_2) \mid a\langle\rangle)$, ale port ***a*** již v procesu existuje v části $(b\langle\rangle.R_1 + a\langle\rangle.R_2)$ proto je nutné nově vytvářenou zprávu odlišit a změnit název nově vytvářené zprávy v první části procesu.

Druhým bodem je libovolná změna pořadí operací výběru. Z výběru je vždy vybírán pouze jeden prvek, nezáleží tedy na pořadí, ve kterém jsou prvky zapsány. $P+Q+R \equiv Q+P+R \equiv P+R+Q$ popisuje ekvivalenci jednotlivých zápisů.

Třetí pravidlo popisuje ekvivalenci souběžného spouštění procesů. První

spuštění procesu P s prázdnou operací je shodné se spuštěním samotného procesu P. Poslední dvě pravidla tohoto bodu popisují ekvivalenci zápisů a chování při výměně pořadí zapsaných procesů nebo jejich prioritizaci uzávorkováním.

Ve čtvrtém pravidle je popsáno chování při vytváření nových zpráv. První bod popisuje ekvivalenci vytváření zprávy x v procesech P a Q. Když není x v procesu P, může být vytvořeno x pouze v procesu Q. Druhý bod ukazuje ekvivalenci vytváření zprávy pro prázdnou operaci a možnost prohození vytvářených zpráv pro proces P.

Posledním pravidlem je možnost přepsání replikace do ekvivalentního tvaru skládajícího se ze souběžného spuštění procesu P, první je získán z replikace, druhý může být kdykoliv až nekonečně mnohokrát replikován.

V případě, že můžeme bez omezení provádět všechny body strukturální shody, jsou procesy sktrukturálně shodné.

2.6. Spojování

V některých případech můžeme požadovat řetěz více procesů za sebou, kdy je pravý port prvního procesu spojen s levým portem následujícího. Tento proces by mohl vypadat takto

B(l,r) = l(x) . r<x> . B(l, r)

to znamená, že levým portem l přijmu zprávu x a následně pravým portem r odešlu x a aktivuji celý proces znovu. Samotný řetěz procesů bude definován takto

B(n) = B¯B¯...¯B, kde počet procesů B je n. Samotné spojení je naznačeno pomocí ¯ a je definováno následovně $P \bar{} Q = new\, m(\{m/r\}P|\{m/l\}Q)$, tato definice říká, že pravý port r prvního procesu a levý port r následujícího procesu jsou přejmenovány na m, což je odesílaná a přijatá zpráva.

2.7. Jednoduché systémy

Jednoduché systémy jsou ty systémy, které splňují následující tři podmínky

$P \equiv new\, \vec{z}\,(M_1 \mid ... \mid M_m \mid !N_1 \mid ... \mid !N_n)$ - standardní forma (1)

- Proces P je strukturálně shodný se standardní formou, ve které jsou všechny replikované komponenty sloučením.
- Podprocesy M, N procesu P neobsahují replikaci. (!)
- Podprocesy M N procesu P neobsahují kompozici (|)

2.8. Datové typy

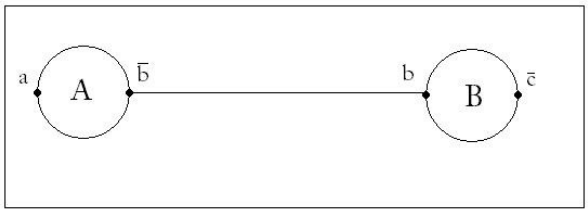
Většina jazyků používá datové typy jako celá čísla, řetězce a jiné, i tyto datové typy je pi-kalkul schopný zaznamenat pomocí proměnných, dvojtečky a jakéhokoliv běžně známého či vymyšleného datového typu. Tyto typy mohou být upřesněny následujícím způsobem: datovým typem, šipkou a bližší specifikací tohoto datového typu.

$$b : BOOL , t : TRUE , c : CONS , l : LIST$$
$$BOOL \rightarrow TRUE , FALSE$$
$$LIST \rightarrow NIL , CONS$$
$$CONS \rightarrow VAL , LIST$$
$$TRUE \rightarrow \varepsilon$$

2.9. Flowgraf

Flowgrafy jsou používány k znázornění komunikujících procesů v daný okamžik a díky toho je nutné při každé změně komunikace vytvořit nový flowgraf. To je velice časově náročné a pracné, proto se flowgrafy používají jen zřídka.

Ve flowgrafech se znázorňují jednotlivé procesy kružnicí s názvem procesu. Komunikační porty jsou znázorněny pomocí plných teček s názvem u procesu. Pro zjednodušení se často nepíšou názvy portů přímo k procesu, ale nad komunikační kanály (hrany) mezi procesy. Na obrázku je zobrazený jednoduchý flowgraf dvou



Ilustrace 3: Jednoduchý flowgraf

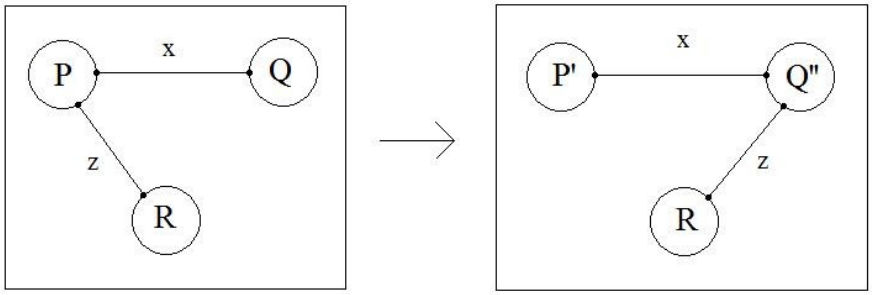
procesů A a B, proces A má porty a, b, proces B má porty b, c.

Příklad procesu zapsaného pi-kalkulem a zobrazeného flowgrafy.

Vytvoří se nové spojení z mezi procesy P a R.

Proces P odešle své spojení z pomocí x procesu Q, pokračuje procesem P'

Zároveň s procesem P vykonává svou činnost proces Q, který přijme zprávu y po x, přepíše všechny výskyty y na z a pokračuje procesem Q'.



Ilustrace 4: Změna ve flowgrafech při přechodu

$$new\, z(P|R) \mid Q \rightarrow P' \mid R \mid \{z/y\}Q'$$
$$P = \bar{x}\langle z\rangle.P'$$
$$Q = x(y).Q'$$

3. Často používané nástroje

V průběhu zpracování celé diplomové práce jsem velice často využíval například bezkontextovou gramatiku a regulární výrazy, které mi v mnoha ohledech usnadnily práci. V této kapitole popíšu a vysvětlím jejich formální definice, které jsem čerpal ze skript k předmětu Teoretická informatika ([6] regulární výrazy str. 110, bezkontextová gramatika str. 119)

3.1. Regulární výrazy

Regulární výrazy jsou velice často využívaným nástrojem pro vyhledávání charakteristických skupin znaků.

Definice:

Regulárními výrazy nad abecedou Σ rozumíme množinu RV (Σ) slov v abecedě Σ ∪ { ∅, ε +, ,* , (,) } (kde předpokládáme, že ∅, ε,+, ·,* , (,) 6∈ Σ), která splňuje tyto podmínky:

• Symboly ∅, ε a symbol a pro každé písmeno a ∈ Σ jsou prvky RV (Σ); tyto symboly také nazýváme elementárními regulárními výrazy.

• Jestliže α, β ∈ RV (Σ), pak také (α + β) ∈ RV (Σ), (α · β) ∈ RV (Σ) a (α*) ∈ RV (Σ).

• RV (Σ) neobsahuje žádné další řetězce, tedy do RV (Σ) patří právě ty řetězce (výrazy), které jsou konstruovány z ∅, ε a písmen abecedy Σ výše uvedenými pravidly.

(α + β) znaménko plus se používá pro výběr jedné z možností, slovo α nebo slovo β.

(α · β) znaménko · označuje zřetězení, slovo α je ihned následováno slovem β

(α*) znak * představuje kvantifikátor množství výskytu, označuje výskyt ani

jednou až nekonečně mnohokrát. V různých způsobech zápisu jsou používány i jiné kvantifikátor, například ? pro jeden nebo žádný výskyt nebo {0-5} pro označení ani jednou až pětkrát. Pokud není určen kvantifikátor jedná se o výskyt právě jednou.

Pro definování povolených znaků slova použiji zápis [A-Ž0-7], který v tomto případě označuje slovo obsahující jen jeden znak ze znaků A-Ž nebo 0-7.

3.2. Bezkontextová gramatika

Bezkontextová gramatika je formální gramatikou, pomáhající s přepisem vstupu na základě přepisovacích pravidel gramatiky, které jsou ve tvaru $A \rightarrow \beta$, kde A je neterminál a β je řetězec neterminálů a terminálů. Výstupem této gramatiky mohou být pouze řetězce terminálů

Definice:

Bezkontextová gramatika je definována jako uspořádaná čtveřice $G = (\Pi, \Sigma, S, P)$, kde

- Π je konečná množina neterminálních symbolů (neterminálů)
- Σ je konečná množina terminálních symbolů (terminálů), přičemž $\Pi \cap \Sigma = \varnothing$
- $S \in \Pi$ je počáteční (startovací) neterminál
- P je konečná množina pravidel typu $A \rightarrow \beta$, kde
  - A je neterminál, tedy $A \in \Pi$
  - β je řetězec složený z terminálů a neterminálů, tedy $\beta \in (\Pi \cup \Sigma)*$.

Pro zjednodušení zápisu bezkontextová gramatika používá znak |, který označuje výběr. Místo zápisu více pravidel se shodným neterminálem na levé straně napíši tento neterminál jen jedenkrát a pravé strany přepisovacích pravidel stejným způsobem napravo, pouze je oddělíme znakem |.

Původní zápis S -> T.S, S -> T+S, S -> T je možné zkrátit na S -> T.S | T+S | T

4. Pi-kalkul elektronicky

Nutnou podmínkou pro zobrazování pi-kalkulu pomocí flowgrafů jsou vstupní data, jejich zápis a uložení v elektronické podobě, kterými se budu zabývat v této kapitole. Krátce se zmíním i o aplikacích, které jsou schopny pi-kalkul zobrazit.

4.1. Vizualizující programy

Podařilo se mi nalézt pouze dva programy (piviztool, adr2graphs), vizualizující pi-kalkul. Vizualizaci ovšem neprováděly samy o sobě, ale používaly k tomu převod do grafové syntaxe označované jako DOT, která byla následně zobrazena pomocí Graphviz. Aplikaci, která by dokázala samostatně vizualizovat pi-kalkul jsem žádnou nenalezl.

Oba tyto programy používají jednoduchý textový zápis pi-kalkulu a jsou schopny maximálně otevřít či uložit data do txt souborů ani jeden neumí vytvořit například xml dokument.

4.2. XML dokument

XML (Extensible Markup Language) byl vytvořen v zimě 1997, jako univerzální rozšiřitelný značkovací jazyk pro přenos jakýchkoliv dat. Jeho výraznou předností je schopnost strukturalizovat data do stromové struktury a přístup ke konkrétním uzlům tohoto stromu je možný pomocí jazyka XPath. Tyto soubory jsou jednoduše přenositelné. XML je rychle čitelný, rychle se vytváří, práce s ním je jednoduchá, viz. [7]

4.2.1. Struktura XML

V této části popíšu základní prvky xml dokumentů. První informací v dokumentu je vždy popis o jaký dokument se jedná s informací o verzi, například <? xml version="1.0"?>.

Tento řádek je následován počátečním tagem kořenového uzlu, jehož ukončení je také poslední informací celého xml dokumentu. Všechny tagy jsou v dokumentech párové, musí tedy začínat <název uzlu> a končit </název uzlu> mezi tyto tagy je zapisován jejich obsah. Pokud tyto tagy nemají žádný obsah, mohou se zapsat zkráceným zápisem <název uzlu />. Uzly jsou libovolně skladatelné, ale nesmí se křížit. Vznikne-li tag v některém uzlu, uzavření musí proběhnout ve stejném uzlu, ne v potomkovi tohoto uzlu.

Počáteční tagy mohou obsahovat i atributy a jejich hodnoty, zapisují se

následovně <nazev_elementu nazev_atributu="hodnota_atributu" />. Pokud chcete do xml vložit komentáře, dají se vložit mezi znaky <!-- komentar -->.

4.3. Pravidla převodu

Již jsem popsal, jak vypadá XML dokument a jaká je jeho struktura, ale stále chybí vysvětlit, jak bude vypadat samotný převod z pi-kalkulu do XML dokumentu a jaké tagy budou v XML dokumentu použity. Dříve než začnu vysvětlovat samotný převod musím upřesnit formální zápis pi-kalkulu, který je v některých případech nejednoznačný, jedná se například o sekvenci, kterou je možné zapisovat pomocí tečky, mezery nebo spojením akcí dohromady. Po upřesnění formálního zápisu popíšu jednotlivé XML tagy pro všechny operace pi-kalkulu.

4.3.1. Pravidla formálního zápisu

- ve vstupním formálním zápisu musí být prvky sekvence odděleny tečkou, aby nedošlo k náhodnému spojení více portů nebo procesů.
- podobně jako u sekvence budou ve vstupním formálním zápisu u klíčového slova new odděleny nově vytvářené porty a procesy čárkou. Porty a procesy, které new ovlivní budou v kulatých závorkách.
- k sobě logicky patřící skupiny paralelismu nebo výběru (sumy) budou řádně uzávorkovány, aby náhodou nedošlo k jejich špatnému zpracování
- odesílání zpráv je formálně popsáno názvem portu s nadtržením a případně odesílanou zprávou v závorkách <, >. Vzhledem k tomu, že nadtržení se v elektronickém zápisu prakticky nedá vytvořit budu používat vždy jen ostré závorky, které nemusí mít žádný obsah.
- Replikaci je možné zapisovat dvěma způsoby vykřičníkem následovaným odesláním zprávy, přijetím zprávy nebo procesem. Druhým způsobem je vykřičník následovaný závorkou, ve které je obsah replikace.

4.3.2. Pravidla XML dokumentu

- Každý proces bude uzavřen v tagu <process> obsahujícím název procesu, tagy <process-port> s jednotlivými počátečními porty procesu a <process-description> s popisem chování příslušného procesu.
- V xml dokumentech jsou používány špičaté závorky pro označení tagů, v pi-kalkulu jsou bohužel používány stejné závorky pro odesílání zpráv přes porty procesů, proto jsem zvolil v mnoha překladačích používané **<** pro < a **>** pro >.
- Suma a paralelismus jsou si ve formálním zápisu velice podobné, obě operace používají stejný zápis jen se odlišují znaménkem + a |. Zápis v XML dokumenty bude pro obě operace stejný, jen s jinými klíčovými slovy. Hlavní uzel bude začínat a končit <sum> (<par>) a uvnitř tohoto uzlu budou jednotlivé prvky označeny <sum-el> (<par-el>).
- U replikace se používá vykřičník následovaný prvky, ke kterým se vztahuje, tyto prvky vložím do tagu <replication>
- Vytvoření nových prvků označuje slovo new, stejným způsobem nazvu uzel <new>, do kterého vložím seznam prvků v uzlech <new-value> a za ně nový uzel <new-el> obsahující prvky, které budou pomocí new ovlivněny.
- Jednotlivé prvky sekvence, které jsou ve formálním zápisu odděleny tečkou budou v samostatných uzlech <seq-el>.
- Příjem zpráv přes port zůstane v původním tvaru.

4.3.3. Příklady převodu do XML dokumentu

a.R	<seq-el>a</seq-el><seq-el>R</seq-el>
x(y,z)	x(y,z)
x<y,z>	x<y,z>
!R	<replication>R</replication>
!(x.z)	<replication> <seq-el>x</seq-el>

```
                                    <seq-el>z</seq-el>
                                </replication>
R | S                           <par>
                                    <par-el>R</par-el>
                                    <par-el>S</par-el>
                                </par>
R + S                           <sum>
                                    <sum-el>R</sum-el>
                                    <sum-el>S</sum-el>
                                </sum>
new a,b(R)                      <new><new-value>a</new-value>
                                    <new-value>b</new-value>
                                    <new-el>R</new-el>
                                </new>
```

Příklad převodu jednoduchého procesu

PRODEJCE(prodejna, banka, na_skladě, ve_výrobě) = prodejna(specif) . prodejna<účet> . (banka(zaplaceno) | ((na_skladě<specif> . na_skladě(auto)) + (ve_výrobě<specif> . ve_výrobě(auto)))) . prodejna<auto>

```
<process>
    <process-name>PRODEJCE</process-name>
    <process-port>prodejna</process-port>
    <process-port>banka</process-port>
    <process-port>na_skladě</process-port>
    <process-port>ve_výrobě</process-port>
    <process-description>
        <seq-el>prodejna(specif)</seq-el>
        <seq-el>prodejna&lt;účet&gt;</seq-el>
        <seq-el>
            <par>
                <par-el>banka(zaplaceno)</par-el>
```

```
                <par-el>
                    <sum>
                        <sum-el>
                            <seq-el>na_skladě&lt;specif&gt;</seq-el>
                            <seq-el>na_skladě(auto)</seq-el>
                        </sum-el>
                        <sum-el>
                            <seq-el>ve_výrobě&lt;specif&gt;</seq-el>
                            <seq-el>ve_výrobě(auto)</seq-el>
                        </sum-el>
                    </sum>
                </par-el>
            </par>
        </seq-el>
        <seq-el>prodejna&lt;auto&gt;</seq-el>
    </process-description>
</process>
```

4.4. Průběh převodu do XML dokumentu

Základem úspěchu fungování programu pro zobrazení pi-kalkulu pomocí flowgrafů je správný převod pi-kalkulu z textové podoby do XML dokumentu, se kterým bude tento program primárně pracovat.

V dnešní době gigabajtových operačních pamětí, vícejádrových procesorů je nejpomalejší částí zpracování dat přístup k disku, na kterém jsou data uchovávána. Proto jsem zvolil jako první krok načtení kompletního obsahu souboru s formálním zápisem pi-kalkulu před postupným načítáním jednotlivých procesů. Při načítání jsou automaticky odstraňovány mezery, kterých je více za sebou. Řádky komentářů začínajících // jsou také opomíjeny a zahozeny.

Následujícím krokem zpracování je za pomoci regulárních výrazů kontrola správnosti zápisu a smazání komentářů začínajících /* a končících */. Při kontrole program zároveň nalézá indexy počátků popisů jednotlivých procesů na jejichž základě jsou procesy odděleny. K nalezení indexů využívá regulárních výrazů, není tedy nutné, aby byly jednotlivé procesy nějakým způsobem odděleny, klíčovým je zápis ve tvaru název procesu následovaný znaménkem rovná se =. Případně může být

za názvem procesu seznam portů v závorkách. V programu jsou k tomuto účelu použity hlavně tyto regulární výrazy: (opačné lomítko \ označuje následující znak jako tisknutelný)

Název procesu - [A-Ž][a-žA-Ž_0-9]*

Název portu - [a-ž][a-žA-Ž0-9_\-]*

Porty v definici procesu - (\([a-ž][a-žA-Ž0-9_\-]*(,[a-ž][a-žA-Ž0-9_\-]*)*\))?

Definice procesu - název procesu + seznam portů v definici procesu + „=“

Po kontrole a rozdělení procesů přistupuji k samotnému převádění XML dokumentu. Každý proces rozdělím na dvě části pomocí =, první část obsahuje název procesu, popřípadě popis portů v závorkách, druhá část je popisem chování procesu. Název procesu je na základě regulárního výrazu nalezeno a vloženo do tagů <process-name>, případné porty procesu jsou z první části rozděleny pomocí čárek a každý port samostatně vložen do <process-port>.

Definice procesu je připravena na vložení do XML dokumentu, zbývá převést složitější část - popis chování procesu pi-kalkulu. XML dokumenty používají závorky <, > pro začátky a konce tagů, proto jsem na počátku převodu zaměnil tyto závorky za < a >.

Pro převod popisu chování procesu pi-kalkulu do XML dokumentu jsem vytvořil následující bezkontextovou gramatiku s terminálními operátory malé p pro název portu procesu a velké P (oproti zvyklostem) pro název procesu pi-kalkulu. U odesílání zpráv jsou pro názornost použity závorky < a >, které jsou již ve skutečnosti nahrazeny.

Přístup v převodu pomocí bezkontextové gramatiky je výhodný díky své rychlosti, protože celý proces je procházen od začátku do konce pouze jednou a postupně se rozkládá na menší části. Oproti tomu přístup zpracovávající samostatně každou operaci zvlášť, prochází celý proces maximálně tolikrát kolik proces obsahuje operací (v tomto případě pětkrát - new, replikace, suma, paralelismus, sekvence), navíc je nutné pracně ošetřovat prioritizaci operací označenou závorkami.

V zápisu bezkontextové gramatiky jsou použity dvěma různými způsoby znaky |, jednou pro znak paralelismu v druhém případě pro oddělení možností formálního zápisu bezkontextové gramatiky. Pro jejich rozlišení v prvním případě nepoužívám v okolí znaku | žádné mezery, v druhém případě formálního zápisu vkládám před i za znak | dvě mezery.

G = ({S, T, U, V, W, N},{port, proces, +, |, !, (,), <, >, new}, S, R)

R: S -> T|S | T+S | T

T -> (S) | T.T | V | new N | !V | !(S)

N-> W (S) | W (S).T | W (S)+S | W (S)|S

V -> port(W) | port<W> | proces(W)

W -> port | port,W

port - jsou slova začínající malým písmenem, jsou složena z malých písmen, velkých písmen (kromě prvního), čísel 0-9, podtržítka a pomlčky

proces - jsou názvy procesů začínající velkým písmenem, mohou být složena z velkých i malých písmen, čísel 0-9, podtržítkem a pomlčkou.

Průběh zpracování neterminálu S:

Počáteční neterminál ***S*** obstarává vytváření výběru a paralelismu. V celém vstupním řetězci hledá první znak sumy nebo paralelismu (bere v úvahu uzávorkování části označené neterminálem ***T***, i když s ní pracuje až v následujícím kroku). Pokud nalezne + nebo | vytvoří pouze v prvním cyklu uzel <par> nebo <sum> a do něj vloží tag se zpracováním neterminálů ***T*** a ***S*** ve tvaru <par-el>***T***</par-el>***S*** nebo <sum-el>***T***</sum-el>***S***. Neterminál ***S*** od teto chvíle vytváří jen uzly <par-el> nebo <sum-el>.

Když neterminál ***S*** nenalezne žádný znak výběru nebo souběžného průběhu, pokračuje neterminálem ***T***.

Průběh zpracování neterminálu T:

Neterminál ***T*** v této gramatice rozlišuje nejvíce operací - sekvenci, replikaci, vytváření nových portů pomocí klíčového slova new a uzávorkování. Když nalezne na začátku řetězce znak replikace ***!***, zkontroluje následující znak, tím rozhodne zda bude pokračovat neterminálem ***V*** nebo ***S***, které vloží do uzlu replikace <repl>.

V případě prvních znaků řetězce v podobě textu ***new*** následovaného mezerou označujících vytváření nových zpráv, gramatika z neterminálu ***T,*** pokračuje zpracováním za pomoci neterminálu ***N***.

Když je prvním znakem celého vstupního řetězce závorka ***(*** a posledním znakem je ukončovací závorka ***)*** stejného páru, jedná se o uzavřený proces a neterminál ***T*** pokračuje dál neterminálem ***S***, do kterého pošle řetězec bez těchto závorek.

Pokud gramatika v neterminálu ***T*** stále nenalezla vhodný další postup, zkontroluje gramatika znaky označující sekvenci, kterými jsou tečky, jsou-li v této

úrovni přítomny vytvoří uzly <seq-el> obsahující řetězce vrácené opětovnou aktivací neterminálu ***T*** (pro zamezení rekurzivního vytváření seq-el v každém neterminálu ***T*** je tomuto neterminálu zasílána informace o probíhajícím nebo neprobíhajícím zpracování sekvence).

Když nenastane ani jedna z výše uvedených možností neterminál T pokračuje neterminálem ***V***.

Zpracování neterminálu N

Neterminál ***N*** slouží ke zpracování všech případů vytváření nových portů pomocí klíčového slova ***new*** následovaného mezerou. V případě kdy gramatika pokračuje tímto neterminálem, vytvoří ***N*** nejprve nový uzel <new>, do něj vloží jednotlivé porty uzavřené pomocí tagu <new-value> a následující obsah závorky, získaný za pomoci neterminálu ***S***, vloží do tagu <new-el>.

Není-li řetězec v této chvíli zpracován celý, zkontroluje gramatika následující znak za závorkou a na jeho základě vloží celý nově vytvořený uzel <new> do uzlu <seq-el>, <par><par-el> nebo <sum><sum-el> a pokračuje za pomoci neterminálu ***T*** nebo ***S***.

Zpracování neterminálu V:

Neterminál ***V*** společně s ***W*** jsou prakticky ukončovacími neterminály zpracování vstupního řetězce gramatikou. Jejich účelem je v případě ***V*** vracet řetězce s příjímáním, odesíláním zpráv přes porty procesů nebo aktivace procesu s definovaným seznamem portů. Neterminál ***W*** jen rekurzivně vytváří seznam portů neterminálu ***V*** nebo ***N***.

Příklad převodu pro názornost lehce upraveného již zmíněného procesu Prodejce.

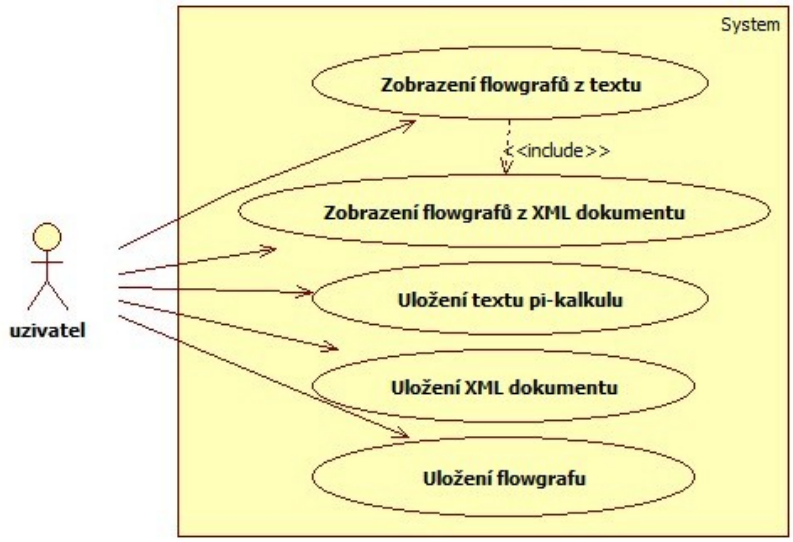
PRODEJCE(prodejna, banka, na_skladě, ve_výrobě) = prodejna<účet> . (banka(zaplaceno) | new specif ((na_skladě<specif> . na_skladě(auto)) + (ve_výrobě<specif> . ve_výrobě(auto)))) . prodejna<auto>)

S-> T -> T.T -> V.T -> port<W>.T -> port<W>.(S) -> port<W>.(T | S) -> port<W>.(V | S) -> port<W>.(port(W)|S) -> port<W>.(port(W)|T) -> port<W>.(port(W)|new N) -> port<W>.(port(W)|new W(S).T) -> port<W>.(port(W)|new W(T+S).T) -> port<W>.(port(W)|new W((S)+S).T) -> port<W>.(port(W)|new W((T)+S).T) -> port<W>.(port(W)|new W((T.T)+S).T) -> port<W>.(port(W)|new W((V.T)+S).T) -> port<W>.(port(W)|new W((port<W>.T) + S). T) -> port<W> . (port(W) | new W((port<W> . V) + S) . T) -> port<W> . (port(W) | new W((port<W> . port(W))+S).T) -> port<W> . (port(W) | new W((port<W>.port(W)) +T). T) -> port<W> . (port(W) | new W((port<W>.port(W))+(S)).T) -> port<W> . (port(W) |new W((port<W>.port(W))+(T)).T) -> port<W>.(port(W)|new W((port<W>.port(W))+(T.T)).T) -> port<W>.(port(W) | new W((port<W>.port(W)) +(V.T)).T) -> port<W>. (port(W)|new W((port<W>.port(W))+(port<W>.T)).T) -> port<W>.(port(W)|new W((port<W>.port(W))+(port<W>.V)).T) -> port<W>. (port(W)|new W((port<W>.port(W))+(port<W>.port(W))).T) -> port<W>.(port(W)| new W((port<W> . port(W)) + (port<W> . port(W))).V) -> port<W> . (port(W)|new W((port<W> . port(W)) + (port<W> . port(W))) . port<W>)

5. návrh programu

Tato kapitola popisuje návrh programu na vykreslení procesů pi-kalkulu pomocí flowgrafů. Detailně se v ní věnuji nosičům dat z XML dokumentu označených jako holdery, které svými hlavními metodami findNext, getNext pomáhající postupnému zpracovávání procesů pi-kalkulu. Hlavní logikou a spojovacím prvkem je metoda findStepsAndCreateInfoDTOs ve třídě Crawler procházející možné kroky pi-kalkulu, které jsou výstupem metody holderů findNext. FindStepsAndCreateInfoDTOs také vytváří datové nosiče flowgrafů upravované metodou holderů getNext. Pro přiblížení systému používám diagram případů užití, třídní diagram a sekvenční diagram jazyka UML. [8]

5.1. Případy užití



Ilustrace 5: Případy užití systému

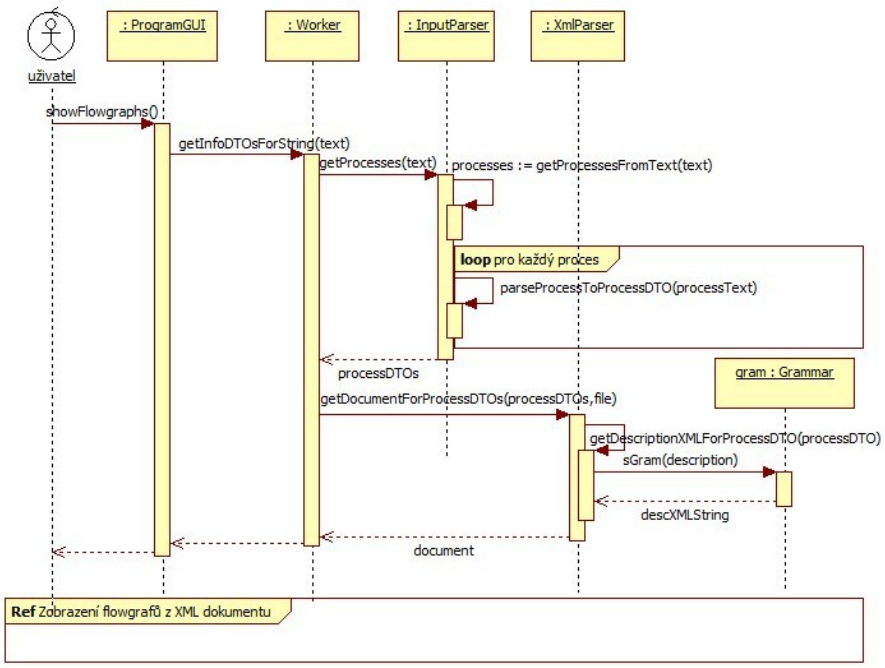
Navrhovaný program je určen pro kohokoliv bez rozdílu věku, povolání nebo práv, proto má pouze jednoho aktéra označeného jako uživatel. Pro jeho potřeby je vytvořeno pět možných případů užití - zobrazení flowgrafů z formálního zápisu pi-kalkulu, uložení pi-kalkulu ve formálním tvaru, zobrazením flowgrafů z XML dokumentu, uložení XML dokumentu nebo poslední uložením obrázku flowgrafu.

Jak již samotný název napovídá zobrazení flowgrafů z textu nebo XML dokumentu jsou si velice podobné, dokonce na sebe navazují, neboť tento program primárně převádí XML dokumenty a ty zobrazuje jako flowgrafy. Zobrazení flowgrafů z textu pouze převádí formální zápis pi-kalkulu na XML dokument, který se dál zpracovává stejným způsobem jako případ užití zobrazení z XML dokumentu.

V případě, že uživatel pracuje s programem, upravuje a zobrazuje formální zápis pi-kalkulu s flowgrafy, může mít požadavek na systém k uložení aktuálního formalismu v podobě textové zápisu, XML dokumentu nebo aktuálního flowgrafu a systém uživateli tyto možnosti poskytne.

5.2. načtení a převod do xml

Tento program je v první řadě určen pro zobrazování flowgrafů z XML dokumentů, které obsahují strukturovaný formální zápis pi-kalkulu. Zobrazování jen na základě XML dokumentů bez možnosti převodu formálního zápisu do podoby těchto dokumentů je velice nepraktické a také nevhodné, neboť nutí uživatele k používání dvou nástrojů prvního na převod do XML formátu a druhým je tato aplikace pro zobrazení procesů pi-kalkulu pomocí flowgrafů. Pro tyto účely jsem vytvořil algoritmus, který pi-kalkul převádí formalismus do adekvátního XML dokumentu.



Ilustrace 6: Zobrazení flowgrafů z textu

Na výše uvedeném obrázku je zobrazen sekvenční diagram s průběhem ukazujícím převod textové podoby pi-kalkulu do XML dokumentu, který po svém zpracování pokračuje zobrazením flowgrafů z XML dokumentu.

Průběh zpracování celého požadavku začíná požadavkem uživatele na zobrazení flowgrafů z formálního zápisu pi-kalkulu. Ve druhém kroku grafické rozhraní zavolá metodu třídy Worker getInfoDTOsForString s textem obsahujícím všechny procesy pi-kalkulu. Worker provede dvě akce, nejdříve získá jednotlivé

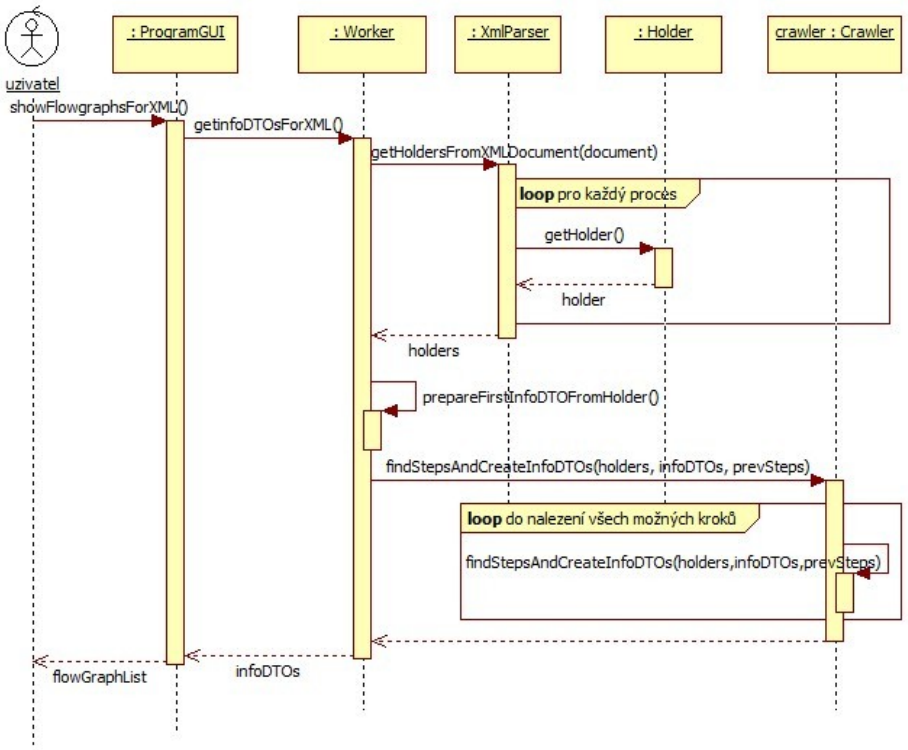
procesy pomocí třídy InputParser, poté vytvoří XML dokument metodami ze třídy XmlParser.

InputParser hledá v textu začátky definic jednotlivých procesů pomocí regulárních výrazů. Jednotlivé procesy od sebe oddělí a vytvoří z nich seznam processDTO. ProcessDTO je datovým nosičem obsahujícím název procesu, jednotlivé porty procesu a popis procesu.

XmlParser převádí postupně za pomoci gramatiky seznam objektů processDTO na ekvivalentní strukturovaný zápis s tagy XML dokumentu, ze kterých vytvoří kompletní XML dokumentu.

V kapitole 4.4 Průběh převodu do XML dokumentu jsem detailně popsal samotný převod a bezkontextovou gramatiku, která je k převodu použita.

5.3. načtení dat z XML do holderů



Ilustrace 7: Zobrazení flowgrafů z XML dokumentu

Abych mohl flowgrafy zobrazit, musím nejprve načíst data z XML dokumentu a vhodně je uložit do samotného programu způsobem, aby nedošlo ke ztrátě dat a hlavně struktury tohoto zápisu. Pro tento účel jsem vytvořil nosiče, které pracovně označuji holdery. Holdery jsou datové nosiče procesů pi-kalkulu s podpůrnými metodami pro práci s pi-kalkulem, detailně jsou popsány v kapitole 5.4 Holdery. Načtení XML dokumentu je možné provést dvěma způsoby, buď je již XML

dokument v paměti počítače díky procesu Zobrazení flowgrafů z textu nebo soubor uživatel otevře pomocí uživatelského rozhraní, které automaticky převede formalismus pi-kalkulu na XML dokument.

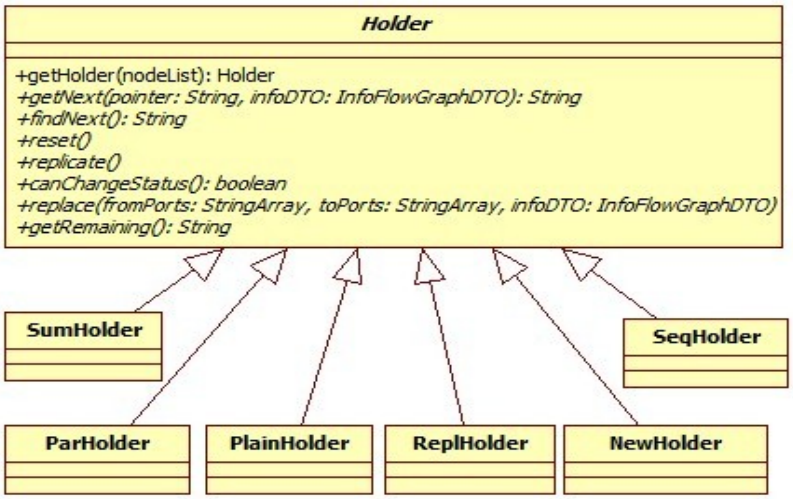
Při načítání XML dokumentu, bylo nutné rozhodnout o způsobu, kterým k dokumentu budu přistupovat. Načtu celý dokument do paměti a budu s ním pracovat jako celkem (DOM XML Parser) nebo budu k jednotlivým uzlům dokumentu přistupovat postupně (SAX XML Parser).

Vybral jsem si přístup DOM, který načte celý strom do paměti a umožňuje cílený přístup k jednotlivým uzlům s možností návratu v dokumentu. Předpokládal jsem, že s tímto dokumentem budu daleko více pracovat, bohužel tato volba nebyla nejšťastnější, neboť jsem nakonec dokument pouze sekvenčně načítal a ukládal do holderů, k čemuž se skvěle hodí právě SAX. Tuto chybu jsem již nestačil opravit.

5.3.1. Průběh načítání XML do holderů

Worker zavolá metodu getHoldersFromXMLDocument, která převádí XML dokumenty na seznam holderů. Ta v uzlu proces-description začne s kontrolou potomků a na základě jejich názvů rekurzivně vytváří na základě názvu uzlu příslušné holdery. Pro uzel <seq-el> vytváří SeqHolder obsahujícího sekvenci, <par> vytváří ParHolder s paralelismem, <sum> SumHolder, <replication> ReplHolder, <new> vytváří NewHolder a posledním holderem je PlainHolder, který obsahuje textový obsah koncového uzlu (listu) stromu XML dokumentu.

5.4. Holdery



Ilustrace 8: holdery

Holdery jsou datovými nosiči formalismu pi-kalkulu, které umožňují zachovat

jeho stromovou strukturu a implementují metody pro práci s těmito daty. Vytváření objektů holderů je za pomoci abstraktní třídy Holder obsahující jednu implementovanou metodu getHolder(Node nl), která na základě argumentu uzlu XML dokumentu <seq-el>, <par>, <sum>, <new>, <repl> nebo obyčejného textu vrací nově vytvořené objekty tříd SeqHolder, ParHolder, SumHolder, NewHolder, ReplHolder nebo PlainHolder pro text. Konstruktorům těchto objektů odesílá potomky aktuálního uzlu a ty rekurzivním způsobem na jejich základě názvu uzlů opět pomocí metody getHolder v abstraktní třídě Holder vytváří další objekty. Tento postup se opakuje až do koncového listu, pro který je inicializován objekt třídy PlainHolder s textem posledního uzlu. Struktura těchto tříd jde vidět na třídním diagramu v obrázku Ilustrace 8: holdery.

Samotná data a struktura holderů nejsou dostatečné pro jejich zobrazení, k tomu je zapotřebí vhodných metod napomáhajících průchodu těchto dat, hledání možných následujicích kroků atd. V této kapitole se pokusím používané metody blíže popsat. Jelikož mohou být třídy do sebe různě zanořeny musí být hlavičky jejich deklarace pro všechny třídy shodné, proto všechny implementují třídu Holder, ve které jsou definovány abstraktní metody - findNext, getNext, reset, replace, replicate, canChangeStatus a getRemaining. Pro uchování informace jaký port a jaká zpráva se po něm posílala slouží ve třídě Holder statická proměnná *sending*, do které se vkládají prázdné zprávy nebo řetězce ve tvaru *názevportu<port,...,port>*.

5.4.1. metoda findNext

Základním smyslem této metody je nalézt všechny další možné kroky pi-kalkulu z aktuálního stavu procesu, které jsou základem interakce s procesy v holderech. Metoda prochází všechny své potomky a vrací řetězec všech možných kroků z aktuálního stavu procesů pi-kalkulu označených složenými závorkami ***{,}*** označující začátek/konec paralelismum, hranatými závorkami ***[,]*** označující výběr a oddělených středníkem ***;*** pro jednotlivé možné kroky v paralelismu nebo výběru. Pro názornost uvedu příklad.

Prodejce = (banka(zaplaceno) | (na_skladě<specif>.na_skladě(auto) +ve_výrobě<specif> .ve_výrobě(auto))) .prodejna<auto>

Strukturu holderů jsem se pokusil znázornit níže, prvním holderem je SeqHolder se dvěma prvky ParHolderem a PlainHolderem (prodejna<auto>), ParHolder obsahuje také dva prvky PlainHolder(banka(zaplacen)), SumHolder. Tento SumHolder obsahuje dva SeqHoldery, v prvním je PlainHolder(na_skladě<specif>), PlainHolder(na_skladě(auto)), ve druhém holdery PlainHolder(ve_výrobě<specif>) a PlainHolder(ve_výrobě(auto)).

SeqHold ParHold PlainHolder banka(zaplaceno)

SumHolder SeqHolder PlainHolder na_skladě<specif>

PlainHolder na_skladě(auto)

SeqHolder PlainHolder ve_výrobě<specif>

PlainHolder ve_výrobě(auto)

PlainHolder prodejna<auto>

Metoda findNext začne svůj průchod v kořenovém holderu SeqHolder, u kterého zkontroluje, který prvek je aktivní a na něj zavolá metodu findNext. Aktivním prvkem je v tomto případě ParHolder, který vloží na začátek a konec výstupního řetězce složené závorky, do středu vkládá výstup metody findNext na všechny své holdery. Tyto výstupy odděluje středníkem. Prvním uzlem ParHolderu je PlainHolder obsahující příjem zprávy přes port banka, metoda findNext PlainHolderu nejdříve zjistí zda má hledat příjem nebo odesílání zpráv podle proměnné v statické proměnné sending ve třídě Holder. Tato proměnná je prázdná, jedná se tedy o hledání odeslání zprávy. PlainHolder vrací prázdný řetězec, protože obsahuje příjem zprávy.

Druhým prvkem ParHolderu je SumHolder, u kterého metoda findNext nejdříve zjišťuje zda již byl SumHolder aktivován, jedná se o první průchod tímto prvkem, proto podobně jako u ParHolderu připraví hranaté závorky na začátek a konec výstupu a pro každý svůj holder zavolá metodu findNext, výstupy oddělí středníkem a vloží mezi závorky. Holdery, které holder výběru v sobě ukládá jsou oba SeqHolder, ty ve svých prvních PlainHolderech naleznou odesílání zpráv po portu na_skladě a celé tyto zprávy vrátí. Koncovým již složeným řetězcem tohoto postupu je „{;[na_skladě<specif>;ve_výrobě<specif>]}“.

Pokud by na začátku celého průchodu metodou findNext bylo uloženo odesílání zprávy přes v proměnné sending třídy Holder, výstupní řetězec by měl stejnou strukturu, ale s jiným obsahem „{banka(zaplaceno);[;]}“, protože metoda findNext bude hledat pouze přijímání zpráv, odesílání nebude brát v potaz. Při odesílání zprávy přes jiný port než banka nebudou v řetězci obsažena žádná slova, pouze závorky a středníky.

Řetězce znaků, které metoda findNext nalézá jsou, po zpracování Crawlerem, základem získávání jednotlivých datových nosičů InfoFlowgraphDTO pro vytvoření flowgrafů metody getNext zpracovávající vybrané kroky.

5.4.2. metoda getNext

Metoda getNext se používá většinou až po nalezení dalších možných kroků crawlerem (5.5 Nalezení všech kroků pi-kalkulu) na základě výstupu metody findNext. Jeden z těchto vybraných kroků metoda getNext provádí. Argumenty metody jsou identifikátor kroku v podobě číselné řady oddělené středníky (0;1;0) a nosič dat InfoFlowgraphDTO, do kterého se mají uložit informace potřebné k vykreslení flowgrafu (5.7 Ukládání dat do InfoProcessDTO).

GetNext vrací řetězec obsahující informaci pro nahrazení portů procesu ve tvaru „<seznam portů na které přepsat>(seznam portů ze kterých přepsat)".

Pokud bylo ve zvoleném kroku vybráno odeslání zprávy přes port, metoda informuje své rodiče o nutnosti zablokování paralelní větve, kterou je tento krok součástí. Není vhodné, aby proces odesílající zprávy stejné zprávy okamžitě následujícím krokem opět dostával.

Při spuštění metody getNext jsou holdery postupně procházeny na základě číselné řady v proměnné pointerString, pokud se jedná o ReplHolder, SeqHolder, NewHolder nebo PlainHolder s tímto řetězcem se nepracuje. Situace se mění v případě holderů paralelismu a výběru. Jedná-li se o paralelismus, je vždy z řetězce odebráno první číslo, které určí kterým prvkem paralelismu se má pokračovat. Následně je smazán středník ze začátku řetězce pointerString a systém pokračuje dál metodou getNext s upraveným řetězcem u potomka aktuálního holderu.

Podobný postup je i u holderu výběru, s jediným rozdílem, volba postupu se dá zvolit pouze jednou a poté si holder již zapamatuje, kterým prvkem má vždy pokračovat. Tato zapamatovaná informace má také vliv na metodu findNext, která nadále u výběru nebere v potaz všechny možnosti, ale pokračuje již zvolenou možností.

V posledním nosiči dat PlainHolderu obsahuje první argument metody getNext prázdný řetězec a začíná se pracovat s druhým argumentem. Pokud PlainHolder obsahuje odeslání zprávy, např. na_skladě<specif>, je do infoDTO zapsána celá informace, po kterém portu a jakou zprávu zasílá. Stejná informace je zapsána i do proměnné sending ve třídě Holder. Do infoDTO systém zapíše informaci o aktuálním procesu, který odesílání provádí. Poslední akcí, kterou metoda getNext v tuto chvíli končí je označení PlainHolderu za použitý a odesláním požadavku na blokaci paralelní větve, jejíž je tento plainHolder součástí.

V případě volby příjmu zprávy systémem PlainHolder načte zprávu z proměnné sending. Tuto zprávu společně se svým obsahem (příjmem zprávy) odešle zpět k přepisu portů. Zároveň se holder označí za použitý a vloží do infoDTO informaci, do kterého procesu byla zpráva odeslána.

5.4.3. metoda canChangeStatus

Při průchodu kompletního holderu i s potomky a hledání nebo výběru následujících kroků není žádoucí procházet uzly, které program navštívil a zpracoval, protože to zpomaluje zpracování. Některé holdery mají mechanismus určující jeho přístupnost. PlainHolder má proměnnou used určující zda již byl nebo nebyl použit holder použit. ParHolder udržuje informace o ukončení uzlu v poli s proměnnými typu boolean, má-li proměnná hodnotu pravda, je příslušný potomek holderu plně zpracován. Jsou-li zpracovány všechny tyto prvky může rodič změnit svůj stav a pokračovat jiným uzlem. SumHolder kontroluje pouze stav zvoleného potomka, když nebyl dosud žádný prvek vybrán, znamená to, že tento holder nebyl ještě zpracováván a metoda canChangeStatus vrací automaticky hodnotu nepravda. Ostatní holdery jen přeposílají výstupy svých potomků.

Tato metoda je důležitá u SeqHolderu, protože ten netuší jak daleko ve zpracování aktuálního uzlu je a zda může pokračovat uzlem následujícím, proto pomocí metody canChangeStatus kontroluje své potomky a dostane-li odpověď pravda, začne pracovat s následujícím uzlem v pořadí.

5.4.4. metoda reset

Metoda reset navazuje na předešlou metodu canChangeStatus. Aktivuje se vždy v případech, je-li nutné se vrátit v práci s holdery o nějaký krok zpět nebo začít úplně znovu. V holderech, které jsem vytvořil není možné žádným způsobem vrátit postup hledání o krok či více zpět, jedinou možností je začít s holdery úplně znova. Což znamená odemknutím všech holderů pomocí metody reset, která zároveň nastavení všechny počáteční hodnoty PlainHolderů do původního stavu.

Důvodem, proč jsem nevytvářel metodu, která by vracela holdery o krok zpět byl problém s efektivním ukládáním jednotlivých přepisů názvů portů po odeslání a přijetí zprávy a s tím spojených velkých zásahů do jednotlivých procesů.

5.4.5. metoda getRemaining

Nastane-li situace, kdy je třeba získat zbývající, ještě nezpracované akce pi-kalkulu jako prostý text, je volána vždy tato metoda. GetRemaining vrací procesy pi-kalkulu v podobě řetězce s příslušnými spojkami pro každou operaci pi-kalkulu. Tato metoda obvykle vrací obsah svých potomků, které upravuje na základě své příslušnosti k holderům, např. seqHolder přidává mezi jednotlivé prvky tečku, SumHolder plus atd. Vyjímkou, která stojí za zmínku je NewHolder, který pokud ještě nebyl aktivován metodou getNext vrací i spojení new s názvy zpráv.

Tato metoda je použita ve všech případech vypisování informací o aktuálním průběhu procesu do textové oblasti aplikace s formálním zápisem pi-kalkulu. Provádí

se okamžitě po aktivaci kroku metodou getNext a zapisuje výstupní řetězec do InfFlowgraphDTO.

5.4.6. metoda replace

Tato metoda slouží jako jediná k přepisu a přidání portů v procesu pi-kalkulu, používá se při vytváření nových portů uzlu new a u přepisu portů po zavolání metody getNext s výběrem příjmu zprávy. Jejími argumenty jsou seznam portů, které mají být zaměněny a seznam portů, kterými mají být nahrazeny. Druhý seznam může být větší a porty jsou do seznamu portů pouze vloženy.

5.5. Nalezení všech kroků pi-kalkulu

Holdery s daty a pomocnými metodami pro práci s nimi jsou připraveny k použití, zbývá rozhodnout jakým způsobem budou procesy pi-kalkulu zpracovávány. Jednou možností je postupný průchod procesy a výběr kroků uživatelem. Druhá volba je automatická a provádí ji aplikace, která hledá všechny možné kroky pomocí holderů a výstupní informace zobrazí jako seznam všech kroků, k jehož položkám je přiřazen formální zápis pi-kalkulu a příslušný flowgraf. V následujících podkapitolách jednotlivé způsoby detailněji rozeberu.

5.5.1. Hledání uživatelem

U mnoha procesů je zbytečné hledat všechny možné způsoby odesílání a přijímání zpráv pi-kalkulu. Tyto procesy na sebe nemusí logicky vůbec navazovat, stroj bohužel není schopen automaticky rozeznat, které akce má přeskočit, a které spustit. Pokud počítač nedokáže rozlišit správnost akce, přichází na řadu uživatel, který ručně vybírá následující kroky.

Uživatel přijde k systému, ten mu zobrazí aktuální procesy pi-kalkulu se zvýrazněnými možnými kroky dalšího postupu a uživatel z těchto kroků vybere další postup. Tento postup se stále opakuje, dokud není uživatel spokojen nebo jsou vyčerpány všechny následující kroky v procesech pi-kalkulu.

Tento program používá druhý způsob a hledá všechny možné kroky, které postupně aktivuje. Ovšem první způsob není těžké vytvořit, stačí k tomu pár kroků. Než uživatel zvolí možnost odeslání nebo příjmu zprávy, je nutné všechny možnosti zvýraznit. K tomu se dá použít metod findNext, která vrátí řetězec s možnými následujícími kroky, ty porovnat s výstupem metody getRemaining a části, které se shodují zvýraznit. Zvýrazněné části procesů pi-kalkulu vložit do komponent umožňujících aktivaci. Po zvolení volby odeslání nebo příjmu zprávy systém zavolá metodu getNext a celou operaci zvýraznění a aktivace kroku opakuje za pomoci uživatele.

5.5.2. Hledání systémem

Nevýhodou této metody je značná náročnost na čas a paměť systému. Výhodou je nalezení všech možností postupu, rychlý přístup k jednotlivým prvkům a zobrazení stavů po jejich nalezení.

Po zvolení způsobu, který bude automaticky hledat všechny možné kroky postupu jsem musel rozhodnout zda budu prohledávat holdery do šířky nebo do hloubky. Rozhodl jsem se pro možnost hledání do hloubky, hlavním důvodem bylo, že výstupem této metody bude seznam kroků, které po sobě logicky následují.

Oproti prohledávání do hloubky u prohledávání do šířky jsou zkoumány postupně prvky po vrstvách a výstupem je seznam všech kroků začínajících veškerými možnostmi z kořenového uzlu následován seznamem všech kroků z jeho potomků, seznam kroků potomků z potomků atd. Pokud by chtěl uživatel u prohledávání do šířky najít krok, který po aktuálním následuje, bylo by to pro první dvě vrstvy poměrně jednoduché, ale od třetí vrstvy téměř „nadlidský" výkon a prakticky by stále hledal, které kroky jsou následující.

Aplikace, kterou jsem vytvořil používá prohledávání do hloubky implementované v metodě findStepsAndCreateInfoDTOs třídy Crawler.

Prohledávání začíná zavoláním metody findNext prvního holderu, zapamatováním všech možných kroků a provedením prvního kroku z tohoto seznamu pomocí metody getNext. V následujícím stavu je rekurzivně volána celá metoda s findNext, getNext až do doby, kdy metoda findNext nenalezne žádný možný krok. V tom případě jsou rekurzivně volané metody postupně ukončovány a jejich stav je ukládán do seznamu datových nosičů InfoFlowgraphDTO s informacemi pro zobrazení flowgrafů.

5.6. Práce s procesy

Zatím jsem rozebíral pouze komunikaci a zpracování dat v holderech obsahujících jen příjem nebo odeslání zpráv. V těchto holderech se mohou objevit i definice procesů a jejich portů, které fungují prakticky jako odkazy na vytvořené holdery s jejich popisem. Pokaždé, když program tento proces nalezne za pomoci metody findNext, začne aplikace zpracovávat příslušný holder nalezeného procesu stejným způsobem jako původní. Než se tak stane, musí příslušný proces existovat a musí být jedinečný pro každý výskyt v PlainHolderu. Nikdy nesmí víc PlainHolderů pracovat se stejným procesem. Tím vzniká problém jak vytvářet a jednoznačně identifikovat procesy, které již jsou/nejsou používány.

Na počátku vyhledávání jednotlivých kroků a získávání informací pro flowgrafy mám seznam holderů, který prohledávám pro nalezení následujících kroků. Tento seznam nechám jako originál pro vytváření ještě neexistujících holderů

a vytvořím si úplně prázdný seznam, do kterého zkopíruji první počáteční holder z originálu. V této chvíli poprvé spouštím metodu findStepsAndCreateInfoDTOs, která hledá, provádí kroky s nově vytvořeným seznamem holderů a vytváří seznam InfoFlowgraphDTO. Pokaždé, když narazí metodou findNext na název procesu provede metoda findNext PlainHolderu tyto kroky:

1. prozkoumá seznam nově vytvořených holderů, zda existuje hledaný holder
2. pokud nenalezl holder procesu v pracovním seznamu holderů, prozkoumá originální seznam a zkopíruje nalezený holder do pracovního seznamu holderů. Když holder nalezne, zavolá jeho metodu findNext.
3. změní jméno nově vytvořeného pracovního holderu na nové, jedinečné
4. stejné jméno nastaví aktivnímu PlainHolderu pro budoucí identifikaci.

Tento postup zajistí aplikaci ponechávání původních procesů bez jakýchkoliv změn k případnému, okamžitému zkopírování a použití. Druhý oddělený seznam holderů je pracovní s daty v různém stádiu zpracování.

Když metoda findNext nalezne proces, prohledá tento proces a jako výstup vrátí řetězec složený identifikátorem procesu následovaný znaménkem rovná se =, za které vloží výstup metody findNext cílového procesu. Celý výstup po zavolání této metody může vypadat následovně: „{;4=[na_skladě<specif>;ve_výrobě<specif>]}“.

5.6.1. Replikace

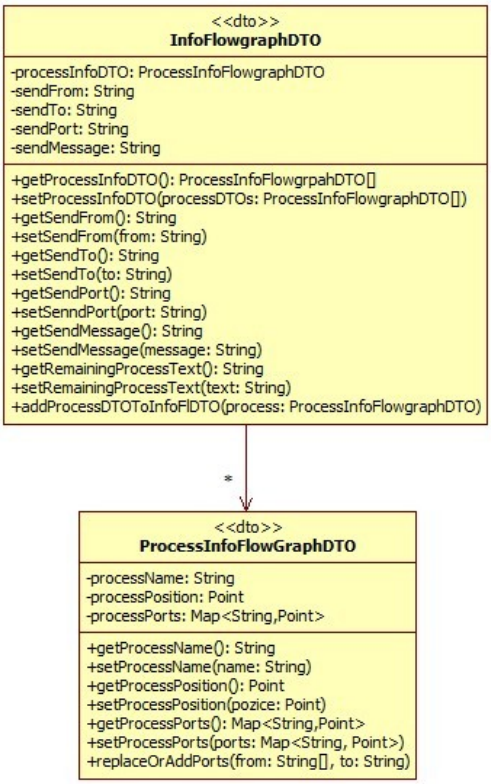
Replikace je specifickým prvkem pi-kalkulu, při replikaci je převáděn replikovaný prvek **!R** na paralelismus sebe sama a shodné replikace **R | !R**, tato operace může být prováděna až nekonečně mnohokrát a díky ní se může celý proces zacyklit a nikdy neskončit. Je tedy nutné vytvořit limitní prvek, počítadlo maximálního provádění replikace, aby se předešlo tomuto nechtěnému efektu.

Tento systém bohužel v této chvíli nepodporuje zpracování replikace, chybí vytvořit metody getNext a findNext holderu ReplHolder, který v této chvíli pouze přeposílá návratové hodnoty svých potomků stejnojmenných metod.

Metoda findNext by měla zůstat skoro stejná jen s úpravou výpisu, při nalezení zjištění, že replikace už byla na tomto prvku replikace provedla maximální počet iterací by vracela pouze prázdný řetězec, tím se zamezí požadování další možné iterace na replikaci metodou getNext.

Metoda getNext změní celou strukturu holderu jehož je součástí vytvořením holderu ParHolder místo stávajícího ReplHolderu, který se přesune do nově vytvořeného ParHolderu a společně s ním se vytvoří nový potomek zpracovávaného ReplHolderu.

5.7. Ukládání dat do InfoProcessDTO



Ilustrace 9: InfoFlowgraphDTO

Nalezení všech možných kroků, které můžeme provést je základem jejich zobrazení. Před každým zavoláním metody getNext holderu v metodě findStepsAndCreateInfoDTOs jsou vytvářeny klony aktuálních datových nosičů InfoFlowgraphDTO, které obsahují název procesů ze kterého a do kterého je zpráva posílána, název portu, po kterém se zpráva posílá a text zprávy. Jsou zde uloženy i další informace v seznamu ProcessInfoFlowgraphDTO, který obsahuje názvy všech zobrazovaných procesů, jejich portů a pozice pro zobrazení. Tyto nosiče dat využívají návrhového vzoru Data transfer object, podle kterého nají na konci názvu zkratku DTO. Provedením kroku getNext je infoFlowgraphDTO pozměněn a připraven k zobrazení.

5.8. Zobrazení flowgrafu

Informace nutné pro zobrazení flowgrafů jsou uloženy v InfoProcessDTO, zbývá je zobrazit. V první řadě je vložen do levého seznamu uživatelského rozhraní programu seznam jednotlivých kroků s pojmenováním ve tvaru procesu, který zprávu posílá, portu, posílané zprávy a cílový proces. Po vybrání položky v seznamu systém

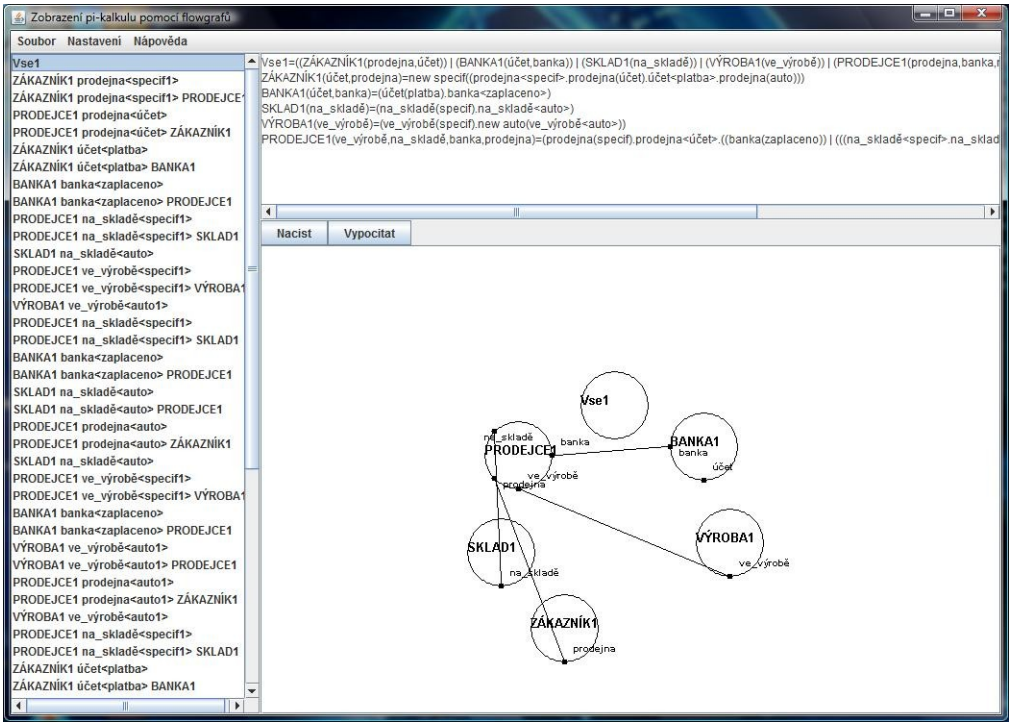
zvolí příslušný InfoProcessDTO, vypočítá pozice pro všechny procesy a jejich porty pro zobrazení na plátno, odešle je třídě FlowgraphCreator, která flowgraph zobrazí.

Zobrazení flowgrafů je konečnou fází zpracování požadavku a odpovídá z části poslednímu bodu zadání této diplomové práce „4. Spolupracujte s diplomantem, který bude zajišťovat zpětný převod ze série flowgrafů do formálního zápisu.“ Spolupráce mezi námi bohužel neproběhla, protože kolega nejevil chuť spolupracovat a díky toho jsem předpokládal, že na své diplomové práci nepracuje. Původní myšlenkou této spolupráce bylo mé použití jeho zobrazovací části a jeho využití mého způsobu ukládání formálního zápisu pi-kalkulu do XML dokumentu. Vzhledem k tomu, že jsem nemohl využít jeho zobrazovací mechanismy, musel jsem vytvořit vlastní, které jsou poměrně jednoduché.

Zobrazovací algoritmus rozloží jednotlivé procesy do kruhu, vytvoří na nich ukazatele portů, s pevným odsazením vypíše jejich názvy. Posledním krokem jejich vykreslení je propojení portů se stejným názvem, pokud je tato možnost v nastavení aktivována.

5.9. Grafické uživatelské rozhraní

Uživatelské rozhraní programu je rozděleno na tři části, přes celou levou stranu je seznam obsahující všechny možné kroky pi-kalkulu, název jednotlivých kroků je ve vytvořen spojením názvu počátečního procesu, názvem portu, přes který je zpráva odeslána, odeslaná zpráva v ostrých závorkách a cílový proces, kterému je zpráva určena.



Ilustrace 10: Grafické uživatelské rozhraní

V horní části obrazovky je upravitelná textová plocha, ve kterém aplikace zobrazuje formální zápis právě používaných procesů pi-kalkulu, ke každé položce seznamu kroků je přiřazen jeden výpis pi-kalkulu. V této části je i tlačítko vypočítat, které zadá příkaz programu pro vytvoření XML dokumentu a nalezení všech možných kroků pi-kalkulu na základě textu v textové ploše.

Poslední třetí částí je plocha v pravém dolním rohu sloužící k zobrazení flowgrafů jednotlivých položek seznamu. Vykreslování není úplně ideální, bohužel chybí sofistikovanější algoritmus, který by rozložil jednotlivé procesy a jejich porty.

6. Závěr

Tato práce se zabývala procesní algebrou Pi-kalkulu, jejím cílem bylo navrhnout a vytvořit softwarový nástroj pro zobrazení procesů pomocí flowgrafů a navrhnout efektivní způsob uchovávání formálního zápisu Pi-kalkulu. Tento převod se podařilo v plné míře vytvořit a je k němu použita cílová struktura XML dokumentu, která je v dnešní době téměř standardem v ukládání dat.

Zobrazení procesů Pi-kalkulu pomocí flowgrafů je z velké části funkční, chybí dokončit implementaci replikace procesů, která je v této práci pouze navržena a kvalitnější algoritmus rozkládající prvky flowgrafu na plátno. Výhodou tohoto programu je, že oproti jiným, které se mi podařilo nalézt, je samostatný a nepotřebuje k funkčnosti žádné externí podpůrné programy.

Z pohledu dalšího vývoje projektu, by bylo vhodné přidat nástroje pro rozpoznávání procesů pi-kalkulu na základě flowgrafů, které měl vytvářet kolega ve své diplomové práci zároveň se mnou. Další možná rozšíření vycházejí z popisu algebry Pi-kalkulu, rozšíření o podporu datových typů nebo zobrazování zřetězení.

Seznam použité literatury

1: Milner, Robin. , Communicating and mobile systems : the Pi calculus, 1999,
2: Joachim Parrow, Handbook of Process Algebra, , http://courses.cs.vt.edu/cs5204/fall09-kafura/Papers/PICalculus/Pi-Calculus-Introduction.pdf
3: Anders Moen, Introduction to Pi-calculus, 2003, http://heim.ifi.uio.no/~andersmo/seminar/foils2003vaar/introPiCalculus.pdf
4: , Wikipedia, π-calculus, 3. 4. 2011, http://en.wikipedia.org/wiki/%CE%A0-calculus
5: L. G. Meredith, Přednášky World News - Pi-calculus, , http://wn.com/Pi-calculus
6: Prof. RNDr. Petr Jančar, CSc., Teoretická informatika - učební text, 2007,
7: Norman Walsh, A Technical Introduction to XML, 1998, http://www.xml.com/pub/a/98/10/guide0.html
8: Arlow J., Neustadt I., UML2 a unifikovaný proces vývoje aplikací., 2007,

Přílohy

1. Zdrojové kódy aplikace – k nalezení na CD v adresáři /implementace.